Progressive load balancing of asynchronous algorithms

Progressive load balancing of asynchronous algorithms

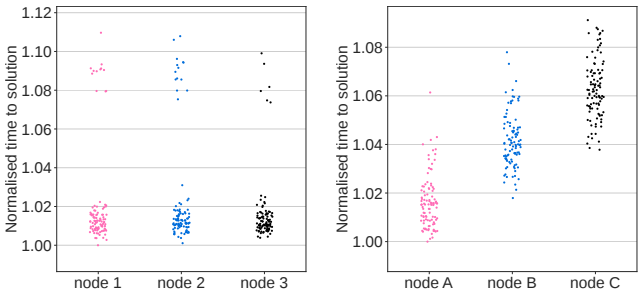
Justs Zarins
University of Edinburgh
Edinburgh, UK
j.zarins@ed.ac.uk

Michèle Weiland
EPCC
Edinburgh, UK
m.weiland@epcc.ed.ac.uk

ABSTRACT

Synchronisation in the presence of noise and hardware performance variability is a key challenge that prevents applications from scaling to large problems and machines. Using asynchronous or semi-synchronous algorithms can help overcome this issue, but at the cost of reduced stability or convergence rate. In this paper we propose progressive load balancing to manage progress imbalance in asynchronous algorithms dynamically. In our technique the balancing is done over time, not instantaneously.

Using Jacobi iterations as a test case, we show that, with CPU performance variability present, this approach leads to higher iteration rate and lower progress imbalance between parts of the solution space. We also show that under these conditions the balanced asynchronous method outperforms synchronous, semi-synchronous and totally asynchronous implementations in terms of time to solution.



(a) ARCHER **(b) Cirrus**

Figure 1: Performance variability within and across different nodes on two HPC machines. Each point is a run of the same application with the same settings.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**;

KEYWORDS

asynchronous algorithm, load balancing, performance variability

ACM Reference Format:
Justs Zarins and Michèle Weiland. 2017. Progressive load balancing of asynchronous algorithms. In *Proceedings of IA^3'17: Seventh Workshop on Irregular Applications: Architectures and Algorithms, Denver, CO, USA, November 12–17, 2017 (IA^3'17)*, 9 pages.
https://doi.org/10.1145/3149704.3149765

1 INTRODUCTION

As supercomputers are growing in size, running large scale, tightly-coupled applications efficiently is becoming more difficult. A key component of the problem is the cost of synchronisation which increases with system noise and performance variability. This affects even high-end HPC machines like ARCHER [1] and Cirrus [2] as shown in Figure 1.

An exciting and promising approach for addressing this problem is to stop enforcing synchronisation points. This results in what are known as "asynchronous" or "chaotic" algorithms [10]; commonly they are iteratively convergent. The cores are allowed to compute using whatever latest data is available to them, which might be "stale", instead of waiting for other threads to catch up. Existing applications of this methodology show good performance and fault tolerance with respect to their synchronous counterparts.

While asynchrony removes the computational cost of requiring all data to arrive at the same time, a different cost takes its place - progress imbalance. This is natural because synchronisation points exist to coordinate progress. An imbalance in progress can result in slower convergence or even failure to converge, as old data is used for updates. This can be countered by putting a strict bound on how stale data is allowed to be, but at a cost to performance.

In this paper we introduce the idea of progressive load balancing – balancing asynchronous algorithms *over time* as opposed to balancing instantaneously. Instead of fine-tuning iteration rates, parts of the working set are periodically moved between computing threads on a node. As a result we limit progress imbalance without adding a large overhead. Our approach is similar to bounded staleness, but it continues to work efficiently in the presence of continuous progress imbalance as a result of hardware performance variability or workload imbalance.

We implement progressive load balancing and use Jacobi's method as an evaluation platform.

Our paper makes the following contributions:

(1) We show that update spread is bounded under a variety of scenarios using progressive load balancing.
(2) We show that, in a shared memory setting, the overhead of balancing is small in most cases.
(3) We show an example of how asynchrony with progressive load balancing is beneficial in terms of time to solution over other synchronisation methods and is successful at minimising the impact of noise.

IA^3'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5136-2/17/11.
https://doi.org/10.1145/3149704.3149765

2 BACKGROUND

Synchronisation in HPC applications is a significant performance bottleneck. This is especially the case in classical bulk synchronous algorithms where progress is made at the rate of the slowest component. Even applications with uniform workload across threads are affected due to hardware performance variation. A machine with homogeneous CPUs and interconnect would still exhibit variable performance due to chip manufacturing differences [3, 17], energy usage management [26], network congestion [15] and OS noise [25]. It is predicted that the issue of performance variability will only grow in future HPC systems [13, 22].

In this context it is attractive to consider asynchronous algorithms. These are normally iterative convergent algorithms. Examples include relaxation methods for linear systems of equations [5, 10], stochastic gradient descend (SGD) [18, 20], finite difference solvers of PDEs [4, 14], adaptive mesh refinement methods [21] and Schwarz methods [23]. Asynchronous algorithms can progress using "stale" values, so one CPU would not have to wait on another that may have stalled, but instead use the most recent value from the stalled worker. However, doing so replaces performance variability with "progress variability", i.e. some parts of the problem space have been progressed towards the solution more than others. In general the time to solution ends up being a function of iteration rate (which increases for asynchronous algorithms) and convergence rate (which may decrease if using stale values for updates). However, too much staleness can result in non-convergence [8].

A solution is to put a bound on how much staleness is allowed. In terms of performance, bounded staleness can tolerate random noise affecting all cores, but would be susceptible to continuous performance variability in the machine (see Section 5).

Thus we are motivated to seek an alternative approach, that would also be resilient against performance variability, via load balancing. In our approach we attempt to exploit the unique property of asynchronous algorithms of tolerating staleness. This allows to load balance in a different way - over time as opposed to instantaneously. Indeed, we do not attempt to *equalise* the iteration rate (or any other load metric), but rather vary it to keep a different metric - progress variation - bounded.

2.1 Related work

One of the earliest formulations of semi-synchronous algorithms was done by Kung [19]. Recently this concept has been applied in machine learning under the name of bounded staleness [12, 16]. In this approach there is a hard limit on how out-of-date values can be before a worker has to wait for fresher ones.

The same idea has been used to implement a memory consistency model, but with the addition of a best effort refresh policy [27]. Refresher threads were used to preemptively refresh stale values. This resulted in a 2.27x speedup over a purely asynchronous implementation.

Bahi et al. have implemented a load balancing algorithm in a 1D stencil application [7]. They found that significant performance gains could be achieved by balancing the iteration rate between components in a grid computing context. Even bigger gains could be gained by using the residual as a load estimator.

Asynchronous algorithms on GPUs face systematic biases in updating due to patterns in GPU thread scheduling [6]. The problem can be tackled by managing the order of execution of thread blocks [11]. This method effectively aims to reduce the staleness of the values used for new updates.

Other approaches accept that there will be significant differences in iteration rates and instead try to manage the negative effects of asynchrony using various algorithmic corrections. For asynchronous SGD examples include tuning algorithmic momentum [24] based on the degree of asynchrony, skipping updates that would direct away from a projected solution [18] or compensating for delayed gradients caused by calculating gradient updates using a stale snapshot of global state [28].

3 METHODOLOGY

In this section we describe our general approach to comparing different synchronisation types of an asynchronous algorithm and the implementation details of progressive load balancing.

3.1 Experimental method

Our general approach is to compare progressive load balanced asynchronous with synchronous, semi-synchronous and asynchronous implementations of the same algorithm. Since there are multiple sources of uncertainty, it is important to take many performance samples, including multiple different nodes, to get a full picture of the range of performance.

Noise generation. Noise can be transient or not present on every node equally (see Figure 1). A variable execution environment makes it difficult to compare experiments. To alleviate this, we inject noise so that there is consistency between experiments and we can discern the effects of different synchronisation types. We simulate a noisy core by running a second, parasite application; for details of the simulation see section 4.3.

Evaluation metrics. We evaluate the different implementations using three metrics:

Iteration rate The number of iterations completed per second. In an asynchronous setting this value is not straightforward, so we define it as the total number of iterations across all threads divided by the number of threads.

Spread A measure of the upper limit of progress imbalance. It is the difference between the maximum and minimum number of updates completed on subdomains of the problem at the end of the run.

Time to solution Time taken to reach a chosen level of accuracy of the solution.

3.2 Progressive load balancing

Our load balancing approach requires that the problem to solve can be split into more parts than there are processing cores. For example, if a 2D iterative stencil application splits the problem domain equally among N CPU cores, we require that each domain is subsplit further on each core. This requirement is not imposing anything new on the application, as it would already have the requirement of domain decomposition in order to parallelise.

Each subdomain has an associated counter to keep track of how many times it has been updated. This information is used by the load balancer to decide which subdomain updating needs to be sped up and which need to be slowed down.

Subdomains start with some initial assignment to threads and are updated as normal. Threads are pinned to 1 core each. At set time intervals a load balancing function is run by one of the threads. This function decides how to reassign subdomains to threads based on differences in the number of updates to subdomains. Due to the coarseness of managing in units of subdomains rather than individual updateable elements, load balancing here is unlikely to result in a stable work distribution where further load balancing is not required. Instead it is continually adjusted so that the progress of subdomains is balanced on average. We believe that progressive load balancing is the best approach to counteract the unpredictable and dynamic nature of system noise.

3.3 Implementation

The load balancing algorithm is detailed in the listing Algorithm 1. Fundamentally, the algorithm works on both shared and distributed memory, however in this paper we present a shared memory implementation as a proof of concept.

The intuition behind the algorithm is as follows: each thread updates a number of subdomains and it is possible to increase or decrease the update (or iteration) rates of those subdomains by removing subdomains from a thread or assigning more to it, respectively. However, reassignment of subdomains to different threads must be done carefully and requires considering the effect this will have on the progress of other subdomains belonging to the affected threads.

In more detail: if a subdomain `topSubd` has had the most updates, the thread that it belongs to, `topThread` is likely fast (e.g. because it is pinned to a core not experiencing any noise or it has less work). The load balancer will reduce the iteration rate of `topSubd` by slowing down thread `topThread` through giving it an additional subdomain to work on. At the same time, we wish to increase the iteration rate of the subdomain `botSubd` that has had the least updates. We find the associated thread `botThread`, which owns `botSubd`, and pick a subdomain from it *other than* `botSubd` and reassign this to `topThread`. Now that `botThread` has one fewer subdomains to update, the iteration rate of `botSubd` will increase.

The iteration rate of the subdomain that is reassigned may go up or down, depending on the relative number of subdomains on `topThread` and `botThread`. The algorithm therefore picks the subdomain that has had the most updates on `botThread` to move to `topThread`, because that subdomain will usually be close to the average in terms of updates completed, so it is unlikely to race too far ahead or fall behind.

Repeatedly performing this load balancing achieves a progressive "braiding" of iteration gradients, hence limiting spread of updates per subdomain (see Figure 2).

These are the user tuneable parameters in the algorithm:

nPairs number of most and least updated subdomains to consider
lowThresh minimum number of subdomains on a thread
highThresh maximum number of subdomains on a thread

```
   Data: nPairs, lowThresh, highThresh
1  doms ← list of subdomains sorted by update count (descending);
2  for i ← 0 to nPairs do
3  |  topSubd ← doms[i];
4  |  botSubd ← doms[−1 − i];
5  |  topThread ← topSubd.GetThread();
6  |  botThread ← botSubd.GetThread();
7  |  if topThread.GetNumSubds() < highThresh and
   |   botThread.GetNumSubds() > lowThresh then
8  |  |  subdToSend ← FindMostUpdatedSubd(botThread);
9  |  |  subdToSend.SetThread(topThread);
10 |  end
11 end
```

Algorithm 1: Progressive load balancing.

The two thresholds (line 7) are to provide some "momentum" and to avoid large swings in iteration rates. Considering more than one pair of subdomains for balancing (line 2) helps the algorithm load balance the whole problem, rather than one part of it. The range of possible settings for these parameters is determined by the degree of subsplitting - itself a tuneable parameter. In general, splitting domains into more subdomains gives greater ability to reduce spread, but it may reduce performance. For example, if the application does stencil computation, performance would decrease due to higher communication frequency between subdomains to exchange halos and less contiguous cache use.

Since the cost of moving data within a CPU or across sockets is different, we developed 3 variants of the algorithm:

Joint All cores are treated the same. Subdomains may be moved across sockets.
Split CPUs on different sockets are balanced separately.
Hybrid CPUs on different sockets are balanced separately. Periodically (this is another tuneable parameter) a random subdomain is moved from a thread on the socket that has completed the least updates on average to a thread on the socket that has done the most.

3.4 Development of the algorithm

The progressive load balancing algorithm was developed iteratively, improving upon a simple starting point. For completeness, we briefly outline our thought process here without entering into too much detail.

Initially all balancing methods worked by moving whole problem domains between threads.

We first tried *swapping* the least and most updated domains between threads. This worked well if all domains took the same amount of time to compute. However, if there was some workload imbalance, the scheme still reduced update spread, but the spread was growing over time.

We then tried a gradient based method, where the most updated domains were moved to threads that had the smallest iteration rate. This method worked better than the previous, even with workload imbalance. Still, we found that increasing workload imbalance in proportion to hardware performance variability would result in a sudden breakdown of the load balancing algorithm.

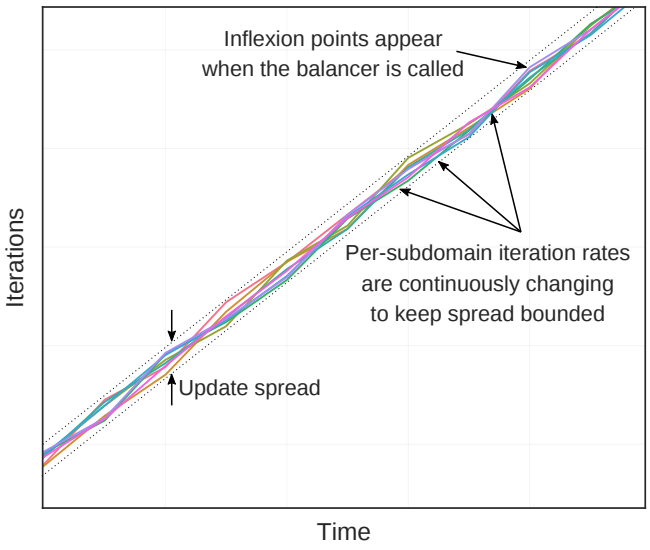


Figure 2: An example of progressive load balancing. Problem subdomains (represented by one coloured line each) are being moved between slow and fast running threads to ensure an overall even progress towards the solution. This picture is drawn using data from a real experiment. Showing only 8 of 96 subdomains for clarity.

Finally, subsplitting problem domains and thinking carefully about the effect of moving subdomains on iteration rates yielded the method presented in this paper. It showed the ability to bound spread with both workload imbalance and hardware performance variation present, and without the requirement to have these in some specific balance.

4 EXPERIMENTS

In this section we describe the specific setup of our environment and experiments used to evaluate the progressive load balancing approach.

4.1 Setup

The experiments were run on 2 actively used HPC systems of different generations. Details can be seen in Table 1.

We ran each experiment a 100 times on 3 nodes making for a total of 300 samples per experiment. The data from the 3 nodes was aggregated. On live HPC systems it would have been extremely time consuming to get the same set of nodes across all the experiments we ran. Instead, we used randomly assigned nodes which resulted in shorter queuing times as well as giving a more representative landscape of performance.

4.2 Test problem and load balancing settings

We chose Jacobi's algorithm in 2D as the test application for our evaluations. It is used to solve the heat equation on a rectangular domain by repeatedly applying a 5-point stencil to average the values in the 4 nearest neighbour cells and store the result in the central cell. When neighbouring cells are located within a different domain, the data is transferred using halo exchange. This algorithm meets asynchronous execution stability requirements [10] which means that we did not have to worry about failure to converge. It is therefore suitable as a comparison point across different synchronisation types, which made it possible to focus on the performance and load balancing aspect of the investigation. While Jacobi's algorithm is not in wide production use, we believe that our findings will be transferable to other stencil applications or different asynchronous algorithms (e.g. ones listed in Section 2).

Table 1: Machine and compilation details.

	ARCHER	Cirrus
System type	Cray XC30	SGI ICE XA
CPU Sockets	2	2
CPU	Intel E5-2697 v2	Intel E5-2695
Core count per CPU	12	18
Clock	2.7 GHz	2.1 GHz
Architecture	Ivy Bridge	Broadwell
L3 cache	30 MB	45 MB
RAM per CPU	32 GB	128 GB
Compiler	CCE 8.5	Intel 16.0
Main compilation flags	Cray default (-O2)	-O2
Noise gen. flags	-O0	-O0

The load balancer is implemented within the Jacobi application using C and OpenMP threading. Threads take turns (round robin) to call the load balancing routine at a set frequency and proceed to reassign subdomains to threads as decided. Threads that are affected by the balancing (either gaining or losing work) are marked as dirty. At the start of an iteration, each thread checks whether its working set has been dirtied before computing updates. If it has, it takes note of the new working set and proceeds calculating updates.

The boundary conditions used were all zeros on 3 sides of the global domain and a Gaussian shaped source on the 4th side. The domains were sized 300 by 300 cells per thread and initialised to 1s. Each domain was subsplit into 4 subdomains; we found this to give a good balance between balancing power and performance overhead. The load balancing lower threshold was set to 2, the upper threshold was set to 6 (a modest offset from the base number of subdomains per thread to avoid large iteration rate swings) and 6 subdomain pairs were considered for moving (see Section 3.2). The latter parameter was empirically found to give reasonable performance in most cases.

The termination criterion for iteration rate and spread experiments was one thread reaching 5000 iterations, or 5000 multiplied by the number of subdomains each domain was split into. For time to solution experiments the criterion was reaching 10^{-4} global l_2-norm of the residual normalised by the initial global l_2-norm of the residual.

4.3 Simulated noise level

In these experiments we simulated only one noisy core. This is a minimum case and illustrates the weakness of bulk synchronicity as the whole node is affected by a single slow thread.

To generate noise we are running a parasite process in the background (due to job scheduling specifics, on ARCHER this is a process and on Cirrus a thread within the main application). The background application (pinned to one core) switches between sleeping for 200 microseconds and performing 10000 iterations of $sum = sum \cdot a + b$. On both machines the loop takes on average 46 microseconds to compute so the noisy core is about 19% slower than the rest. This level of CPU frequency variation can be reasonably expected when applying even a small power cap [17].

In addition to the amount of noise, its noise placement has an effect on the time to solution. Threads are usually pinned to cores in HPC applications because doing so removes time wasted due to thread migration. Given that the problem is decomposed based on threads, a noisy core would affect a particular part of the problem domain. Which domains are sensitive to noise is problem dependent, but, given a complex set of equations, it could be most of the problem space. In our experiments, to demonstrate the effect on convergence rate, we placed the noise next to the boundary that contains the source, as this domain was found to be sensitive to to stale values.

5 EVALUATION

In this section we present experiment results and evaluate and compare the different synchronisation types based on the metrics defined in Section 3.

Figure 3 shows how iteration rate and spread compares across synchronisation types. The best methods are in the lower right corner, i.e. the aim is to minimise spread *and* maximise iteration rate. Also, the the best methods will not change position on the plot by much when noise is added.

Figures 4 and 5 show the time to solution of a representative subset of methods. Less time and smaller variance is better.

The different synchronisation types are denoted using the following labels:

sync synchronised by global barrier

ssync(n) halos from neighbouring domains must be within n iterations of the updating cell

async(n) totally asynchronous version with each domain subsplit into n subdomains

Load balancing types are specified by:

split(f) each CPU socket is balanced independently every f seconds

joint(f) all CPU sockets are balanced together every f seconds

hybrid(f, n) each CPU socket is balanced independently every f seconds and cross socket adjustments are done every nth balancing

5.1 Spread reduction

Adding load balancing to asynchronous Jacobi decreases update spread, with higher load balancing frequency resulting in lower spread (see Figure 3). Invoking the balancer more often results in a tighter "braid", as illustrated in Figure 2, thus reducing spread.

The spread of load balanced versions is comparable to, or lower than, the semi-synchronous versions, except for ssync(1). Among the load balanced versions, the joint scheme achieves the lowest spread. The split and hybrid schemes follow closely, however it should be noted that the split scheme would slowly grow in update spread with increased iteration count while the joint and hybrid schemes would remain steady. This is the case because the split scheme does not exchange subdomains between CPU sockets, so load balancing is done with respect to each socket separately.

Adding noise to a core has the least impact on spread when using a load balanced scheme. Across the two machines, this ranges from a *decrease* in spread of 38% (ARCHER, async(4) + hybrid(0.001,500)) to an increase by 24% (Cirrus, asnyc(4) + hybrid(0.001, 500)). The semi-synchronous schemes increased by between 8% (ARCHER, ssync(1)) and 76% (ARCHER, ssync(30)), while the totally asynchronous schemes range between an increase of 107% (Cirrus, async(1)) to 443% (ARCHER, async(1)).

It is an interesting observation that some of the balancing variants performed better with added noise. We postulate that these variants benefit from iteration rate changes that are less sharp. For example, removing a subdomain from the noisy thread would make the iteration rates increase by a smaller amount than if the subdomains were on a normal thread, thus avoiding overshooting the spread bound.

5.2 Iteration rate

The best performance in terms of iteration rate is achieved by totally asynchronous methods, though ssync(30) on ARCHER is an exception (Fig. 3c).

The overhead (with respect to async(1)) of load balancing varies significantly by method. On Cirrus, split balancing has an overhead of 1%–2%, joint 7%–9% and hybrid 1%–2%. On ARCHER, split balancing has an overhead of 5%–7%, joint 17%–19% and hybrid 4%–7%; it should be noted that the subsplitting itself introduces a 4% overhead on ARCHER (async(4) compared to async(1)).

The joint policy has the worst iteration rate due to data movement across CPU sockets. The hybrid policy mitigates this issue, often providing performance similar to or exceeding the split policy. In all cases increasing load balancing frequency has a negative impact on iteration rate.

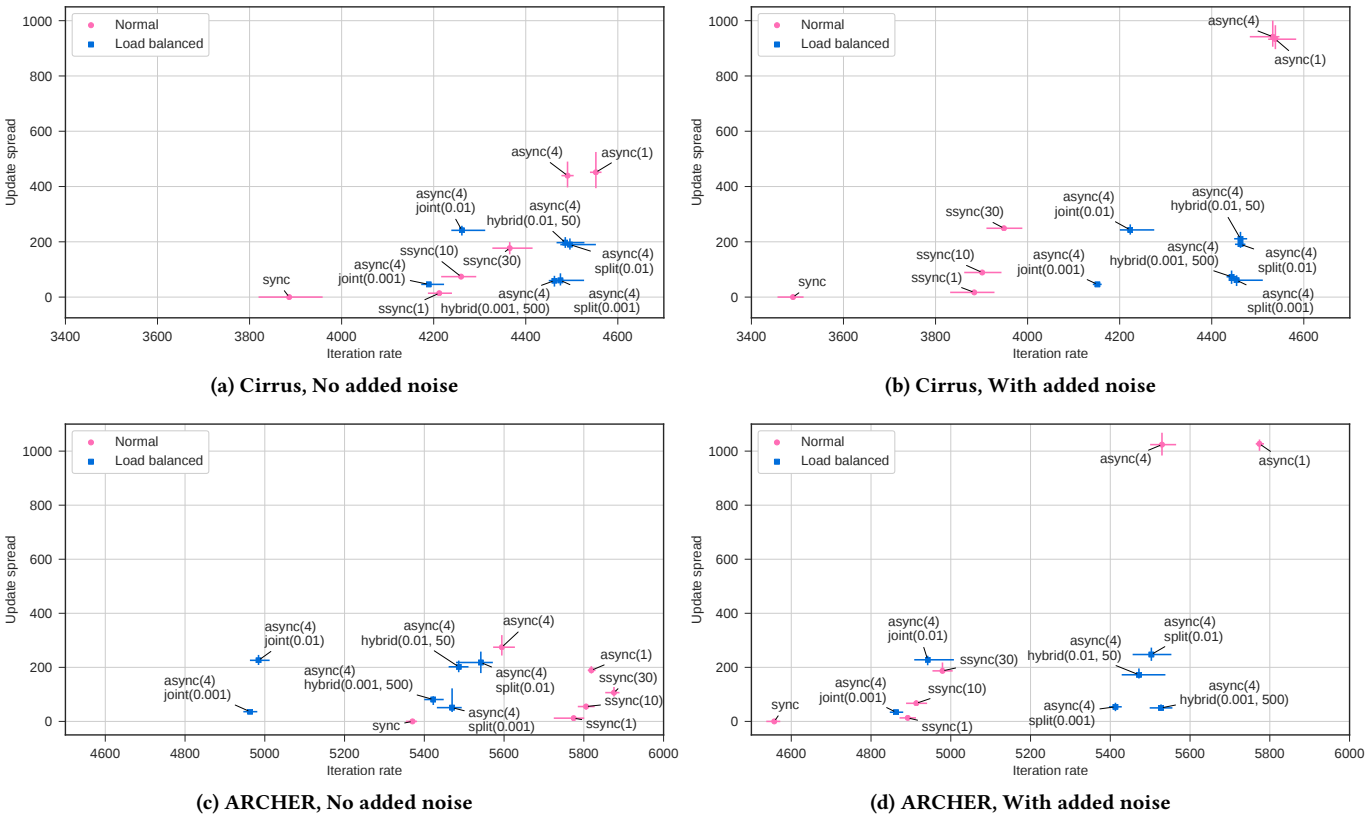
Importantly, the load balanced versions are not heavily affected (less than 1% slowdown on Cirrus, and up to 2% on ARCHER) by the addition of noise - it is effectively spread out among the cores. The addition of progressive load balancing retains the essential property of asynchronous methods to resist noise. Synchronous and semi-synchronous methods are very sensitive to noise and largely become slower (8%–10% on Cirrus, 15% on ARCHER) than load balanced asynchronous variants.

Overall, the relative performance of different synchronisation types is affected by both algorithm settings and machine parameters, as evidenced by the different positions of equal points in the landscape plots (Fig. 3). Nevertheless, the above analysis points out trends that ought to be generalisable and load balancer overhead is low in most cases.

5.3 Time to solution improvement

Figures 4 and 5 give an example of the benefits of the progressively load balanced asynchronous approach.

When the systems are running normally, the asynchronous methods converge in the least time. The version with split load balancing follows close behind. Now, if noise is added to the systems, there



(a) Cirrus, No added noise

(b) Cirrus, With added noise

(c) ARCHER, No added noise

(d) ARCHER, With added noise

Figure 3: Landscape of synchronisation types on Cirrus and ARCHER. The best synchronisation methods are in the bottom right corners (high iteration rate and low spread), and do not change significantly when noise is added. The points represent median values and the error bars show the 25th and 75th percentiles.

is a drastic difference in time to solution response, as shown in Table 2. The synchronous and semi-synchronous versions take the longest to converge; they are limited by the slowest component. The totally asynchronous versions leave behind the slow running thread thus maintaining their iteration rate, but these iterations are less useful due to the increase in update spread. Adding load balancing successfully mitigates the negative effect of the noisy core. The balanced versions effectively distribute the penalty of one slow core across all available cores on the node.

As a result, with hardware performance variability present, our best load balanced method resulted in 22%–25% speedup over synchronous, 14%–19% speedup over semi-synchronous and 5%–8% speedup over totally asynchronous schemes.

Based on the landscapes in Figure 3, we expected the split(0.001) load balanced version to converge the quickest on Cirrus and hybrid(0.001, 500) on ARCHER. The results on Cirrus (Fig. 4) met our expectation, but on ARCHER (Fig. 5) they did not; instead, the split(0.001) balancer gave the quickest convergence again. It appears that update spread, while useful and simple to evaluate, may not be the precise metric to use when choosing the optimum load balancing goal.

Table 2: Time to solution increase of different synchronisation types when adding 19% noise to one core. Cirrus has 36 cores on a node and ARCHER has 24.

	Cirrus	ARCHER
Normal		
sync	13%	18%
ssync(1)	11%	18%
ssync(30)	11%	18%
async(1)	8%	10%
async(4)	9%	11%
Load balanced		
async(4) + split(0.001)	1%	<1%
async(4) + joint(0.001)	1%	1%
async(4) + hybrid(0.001, 500)	-2%	1%

6 FUTURE WORK

Distributed memory. More noise is expected in the distributed memory case when the network and long data transfer latencies come into effect, thus we intend to investigate progressive load

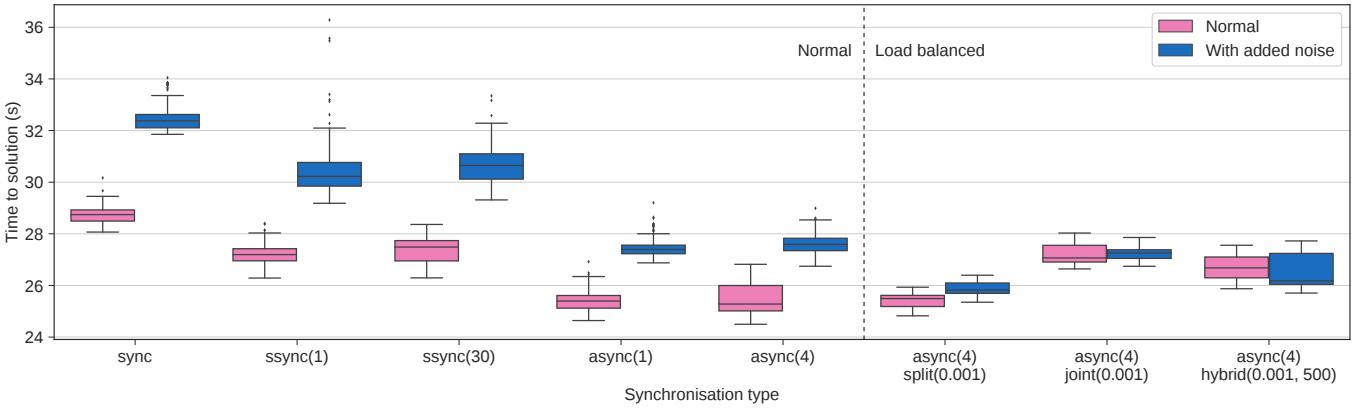


Figure 4: Time to solution on Cirrus. The load balanced versions are least sensitive to noise.

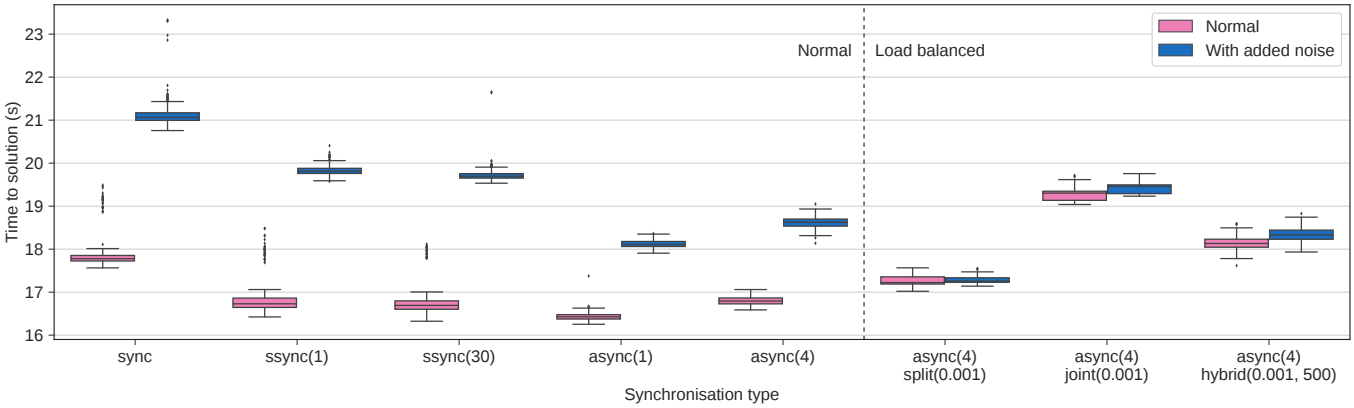


Figure 5: Time to solution on ARCHER. The load balanced versions are least sensitive to noise.

balancing in that setting as well. The hybrid load balancer models an approach that can be taken in distributed memory, as it is concerned with mixing local and global load balancing already, but between CPU sockets rather than nodes. In distributed memory the algorithm would be largely the same but with exchange frequency tweaks to account for longer transfer times. An obvious challenge is whether balancing decisions can be made using global knowledge or would it have to be limited to neighbourhoods and rely on dissipative balancing.

Other applications. All the balancing logic happens within a routine not specific to our Jacobi application, so the balancer could be abstracted out as a library in the future. Given small changes to the user code, this would enable extension to other stencil based asynchronous applications.

We also intend to evaluate progressive load balancing in the context of Stochastic Gradient Descent. This algorithm is widely used in machine learning and can be run asynchronously, however using stale values reduces statistical efficiency. We want to investigate whether balancing the progress rate of learners would lead to convergence rate similar to that of Synchronous SGD while maintaining the hardware efficiency advantage of asynchronous methods.

Comparison with work stealing. A popular load balancing strategy which is similar to our approach is work stealing [9]. It would be interesting to see how this method compares with progressive load balancing. The implementation, however, is not obvious because in principle an asynchronous algorithm always has "available work" because it can continue iterating using the latest available data, even if it is stale, and thus would not have a need to steal work. More research would be required to fully evaluate this load balancing strategy in the context of asynchronous algorithms.

7 CONCLUSIONS

We have presented progressive load balancing – an approach to limit progress imbalance in asynchronous algorithms. Using Jacobi's algorithm as a test case, we have shown that an implementation of this method lowers update spread while maintaining a high iteration rate under most settings, especially in the presence of noise. As a result, our load balanced method achieved a 5%–25% speedup over other synchronisation types, with 19% noise added to one core.

ACKNOWLEDGMENTS

We would like to thank Martin Ruefenacht and Dave Turner for various discussions on the topics covered in this paper.

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (pervasiveparallelism.inf.ed.ac.uk) from the UK Engineering and Physical Sciences Research Council (EPSRC).

This work used the ARCHER UK National Supercomputing Service [1] and EPCC's Cirrus HPC Service [2].

REFERENCES

[1] 2017. ARCHER. (2017). Retrieved 5-9-2017 from http://www.archer.ac.uk

[2] 2017. Cirrus. (2017). Retrieved 5-9-2017 from https://www.epcc.ed.ac.uk/cirrus

[3] Keith A. Bowman, Steven Duvall, and J.D. Meindl. 2002. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits* 37, 2 (Feb 2002), 183–190.

[4] Dganit Amitai, Amir Averbuch, Samuel Itzikowitz, and Eli Turkel. 1994. Asynchronous and corrected-asynchronous finite difference solutions of PDEs on MIMD multiprocessors. *Numerical Algorithms* 6, 2 (sep 1994), 275–296. https://doi.org/10.1007/BF02142675

[5] Hartwig Anzt, Stanimire Tomov, Jack Dongarra, and Vincent Heuveline. 2013. A block-asynchronous relaxation method for graphics processing units. *J. Parallel and Distrib. Comput.* 73, 12 (dec 2013), 1613–1626. https://doi.org/10.1016/j.jpdc.2013.05.008

[6] Hartwig Anzt, Stanimire Tomov, Jack Dongarra, and Vincent Heuveline. 2013. A block-asynchronous relaxation method for graphics processing units. *J. Parallel and Distrib. Comput.* 73, 12 (dec 2013), 1613–1626. https://doi.org/10.1016/j.jpdc.2013.05.008

[7] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. 2005. Dynamic Load Balancing and Efficient Load Estimators for Asynchronous Iterative Algorithms. *IEEE Trans. Parallel Distrib. Syst.* 16, 4 (April 2005), 289–299. https://doi.org/10.1109/TPDS.2005.45

[8] Dimitri P Bertsekas and John N Tsitsiklis. 1991. Some aspects of parallel and distributed iterative algorithms – a survey. *Automatica* 27, 1 (1991), 3–21.

[9] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[10] D. Chazan and W. Miranker. 1969. Chaotic relaxation. *Linear Algebra Appl.* 2, 2 (apr 1969), 199–222. https://doi.org/10.1016/0024-3795(69)90028-7

[11] Edmund Chow and Hartwig Anzt. 2015. *Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs*. Lecture Notes in Computer Science, Vol. 9137. Springer International Publishing. https://doi.org/10.1007/978-3-319-20119-1

[12] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. 2013. Solving the Straggler Problem with Bounded Staleness. In *Proceedings of The 14th Workshop on Hot Topics in Operating Systems*.

[13] Jack Dongarra et al. 2011. The International Exascale Software Project Roadmap. *Int. J. High Perform. Comput. Appl.* 25, 1 (Feb. 2011), 3–60. https://doi.org/10.1177/1094342010391989

[14] Diego A. Donzis and Konduri Aditya. 2014. Asynchronous finite-difference schemes for partial differential equations. *J. Comput. Phys.* 274 (oct 2014), 370–392. https://doi.org/10.1016/j.jcp.2014.06.017

[15] Pedro J Garcia, J Flich, J Duato, I Johnson, Francisco J Quiles, and F Naven. 2005. Dynamic evolution of congestion trees: Analysis and impact on switch architecture. *Lecture Notes in Computer Science (HiPEAC 2005)* 3793 (2005), 266–285.

[16] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems*. 1223–1231.

[17] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, et al. 2015. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 78.

[18] Janis Keuper and Franz-Josef Pfreundt. 2015. Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 1.

[19] H T Kung. 1976. Synchronized and asynchronous parallel algorithms for multiprocessors. *New Directions and Recent Results in Algorithms and Complexity* (1976).

[20] Thorsten Kurth, Jian Zhang, Nadathur Satish, Ioannis Mitliagkas, Evan Racah, Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. 2017. Deep Learning at 15PF: Supervised and Semi-Supervised Classification for Scientific Data. *arXiv preprint arXiv:1708.05256* (2017).

[21] Barry Lee, Stephen F. McCormick, Bobby Philip, and Daniel J. Quinlan. 2003. Asynchronous Fast Adaptive Composite-Grid Methods: Numerical Results. *SIAM Journal on Scientific Computing* 25, 2 (jan 2003), 682–700. https://doi.org/10.1137/S1064827502407536

[22] Robert Lucas, James Ang, Keren Bergman, and Shekhar Borkar. 2014. DOE ASCAC Subcommittee Report February 10, 2014. (2014).

[23] Frédéric Magoulès, Daniel B. Szyld, and Cédric Venet. 2017. Asynchronous optimized Schwarz methods with and without overlap. *Numer. Math.* 137, 1 (01 Sep 2017), 199–227. https://doi.org/10.1007/s00211-017-0872-z

[24] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. 2016. Asynchrony begets momentum, with an application to deep learning. In *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*. IEEE, 997–1004.

[25] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. 2003. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 55–55.

[26] Allan Porterfield, Rob Fowler, Sridutt Bhalachandra, Barry Rountree, Diptorup Deb, and Rob Lewis. 2015. Application runtime variability and power optimization for exascale computers. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 3.

[27] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, 861–878. https://doi.org/10.1145/2660193.2660227

[28] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. 2017. Asynchronous Stochastic Gradient Descent with Delay Compensation. In *International Conference on Machine Learning*. 4120–4129.

A ARTIFACT DESCRIPTION

A.1 Abstract

We provide code and scripts to reproduce the same experiments which were used in evaluation of our proposed method - Progressive load balancing. The artifact are meant to provide evidence for the contributions of the paper by producing similar results to the ones published. We also provide the original data presented in the paper.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithm:** Progressive load balancing.
- **Run-time environment:** Linux. Require C compiler with MPI and OpenMP support.
- **Hardware:** Shared memory node with 1 or 2 sockets and Intel CPUs, preferably a recent Xeon with a total of 12 or more cores in the node.
- **Execution:** Sole user of node, process and thread pinning, preferably a batch submission system like PBS.
- **Output:** Figures and table from paper.
- **Experiment workflow:** Customise Makefile and batch submission script. Use provided bash and python scripts to generate experiments used in the paper and submit them to back end node. Use python script to generate figures.
- **Experiment customization:** Edit experiment configuration file and submission script.
- **Publicly available?:** Yes.

A.2.2 How delivered. Download code from public git repository at `https://bitbucket.org/Justs/ia3_2017.git`

A.2.3 Hardware dependencies. Shared memory node with 1 or 2 sockets and Intel CPUs, preferably a recent Xeon with a total of 12 or more cores in the node.

A.2.4 Software dependencies.

- Linux
- C compilation environment with support for OpenMP and MPI
- python 2
- python 3 with modules:
  - matplotlib
  - seaborn
  - numpy
  - pandas

A.3 Installation

Edit Makefile to use your preferred C compiler and link with MPI and OpenMP. Run `make` to compile.

A.4 Experiment workflow

There are detailed instructions and examples in `README` files in the root and experiments folders. The workflow is as follows:

(1) Use the provided submission scripts and `README` in the root folder to produce your own submission script, with thread/process pinning appropriate for your environment.
(2) Navigate to the folder `experiments`.
(3) Use the `README` in the folder `experiments` to augment your submission script for experiment output redirection.
(4) Use the script `generate.py` to generate experiments used in the paper.
(5) Launch experiments using the script `runBatch.sh`.
(6) Run `bash generate_plots.sh` to summarise experiments as figures.

A.5 Evaluation and expected result

The final outputs are 3 pdf figures and 1 table, corresponding to the ones in the Evaluation section of the associated paper. The results likely will not match the ones presented in our paper exactly, but we expect the trends to be the same. In particular:

- **Landscapes:**
  - The update spread increases significantly (moves up on y axis) for async methods without load balancing when noise is added.
  - The iteration rate decreases significantly (moves left on x axis) for sync and ssync method when noise is added.
  - The load balanced async methods do not move significantly when noise is added. Additionally, they are in bottom right corner of the landscape plot (i.e. spread is low and iteration rate is high (in most cases). (paper contributions 1 and 2)
- **Boxplot:** Some of the load balanced version are fastest when noise is added. (paper contribution 3)
- **Table:** Load balanced versions experience least slowdown when noise is added. (paper contribution 3)

A.6 Experiment customization

New experiments can be created by following the pattern of the files in the `experiments` folder. All algorithm settings are set in the application's input file `config.txt`. The meaning of the variables is explained in the `README` in the root folder.

A.7 Notes

The original data is available as a separate download; see instructions at the end of `experiments/README`.